John von Neumann Institute for Computing

Automatic Computation of Sensitivities for a Parallel Aerodynamic Simulation

Arno Rasch, H. Martin Bücker, Christian H. Bischof

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 303-310, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Automatic Computation of Sensitivities for a Parallel Aerodynamic Simulation

Arno Rasch, H. Martin Bücker, and Christian H. Bischof

Institute for Scientific Computing
RWTH Aachen University, D–52056 Aachen, Germany
E-mail: {rasch, buecker, bischof}@sc.rwth-aachen.de

Derivatives of functions given in the form of large-scale simulation codes are frequently used in computational science and engineering. Examples include design optimization, parameter estimation, solution of nonlinear systems, and inverse problems. In this note we address the computation of derivatives of a parallel computational fluid dynamics code by automatic differentiation. More precisely, we are interested in the derivatives of the flow field around a three-dimensional airfoil with respect to the angle of attack and the yaw angle. We discuss strategies for transforming MPI commands by the forward and reverse modes of automatic differentiation and report performance results on a Sun Fire E2900.

1 Introduction

An interdisciplinary team of engineers, mathematicians, and computer scientists at RWTH Aachen University is currently bringing together computational fluid dynamics with experimental data for high-lift and cruise configurations of civil aircraft in the full range of Mach and Reynolds numbers. The aim of the work carried out within the context of the collaborative research center SFB 401 "Modulation of flow and fluid–structure interaction at airplane wings" is to better understand the dynamics of the vortex system which is generated by large aircraft during take-off and landing. Such a detailed knowledge of the wake flow field is essential to estimate safe-separation distances between aircraft in take-off and landing.

Developed at the Institute of Aerodynamics, the TFS package[1,2] solves the Navier–Stokes equations of a compressible ideal gas in two or three space dimensions with a finite volume formulation. The spatial discretization of the convective terms follows a variant of the advective upstream splitting method ($AUSM^+$) proposed by Liou.[3,4] The viscous stresses are centrally discretized to second-order accuracy. The implicit method uses a relaxation-type linear solver whereas the explicit method relies on a multigrid scheme. A serial as well as a block-oriented MPI-parallelized version of TFS based on domain decomposition are available.

Given the TFS code, we are interested in a qualitative and quantified analysis of the dependences of certain program outputs on certain input parameters. Such information is invaluable for optimizing objective functions with respect to suitable decision variables or for the identification of model parameters with respect to given experimental data. For instance, aircraft design[5] or analysis of turbulence parameters[6] require the sensitivities of functions given in the form of large-scale Navier–Stokes solvers. A recent trend in computational science and engineering is to compute such sensitivities by a program transformation known as automatic differentiation (AD).[7,8]

In the present work, we are interested in the derivatives of the flow field with respect to the angle of attack and the yaw angle, making it necessary to transform a parallel program. No current software implementing the AD transformation is capable of fully handling MPI or OpenMP parallel code. So, it is no surprise that there is hardly any publication in the open literature describing the application of AD to a nontrivial parallel code, an exception being an article by Heimbach et al.[9] The contribution of this work is to sketch the underlying ideas of transforming a parallel code by AD and to demonstrate the feasibility of that approach on a concrete real-world problem arising from aerodynamics.

In Section 2, we briefly explain the automatic differentiation technique and its application to the (serial) TFS code. The transformation of the constructs involving MPI-based parallelism is addressed in Section 3. Performance results of the AD-generated parallel program are reported in Section 4.

2 Automatic Differentiation of TFS

Automatic differentiation (AD) refers to a set of techniques for transforming a given computer program P into a new program P' capable of computing derivatives in addition to the original output. More precisely, if P implements some function

$$f : \mathbb{R}^n \to \mathbb{R}^m, \tag{2.1}$$

mapping an input x to an output $y = f(x)$, the transformed program P' computes not only the function value y but also the $m \times n$ Jacobian matrix $J = \partial y / \partial x$ at the same point x. The AD-generated program P' is called the *differentiated* program of the *original* program P.

Throughout this note, we consider the computation of the flow field for a given angle of attack and yaw angle using TFS. From a conceptual point of view, this computation is carried out by a routine

```
TFS(alpha,beta,vel,pre,...)
```

where the scalar input variables `alpha` and `beta` represent the angle of attack and the yaw angle, respectively. The arrays `vel(1:3*L)` and `pre(1:L)` denote the velocity and pressure at L grid points in three space dimensions. So, we consider the TFS code to implement some function (2.1) with $n = 2$ and $m = 4L$ computing velocity and pressure from two angles.

Specifying that one is interested in the derivatives of the variables `vel` and `pre` with respect to the variables `alpha` and `beta`, automatic differentiation with the ADIFOR[10] tool then transforms the original `TFS` code into a new code

```
g_TFS(g_alpha,alpha,g_beta,beta,g_vel,vel,g_pre,pre,...)
```

where, in addition to all variables occurring in the parameter list of `TFS`, new derivative variables with the prefix `g_` are introduced. These new derivative variables are designed to store two scalar derivatives for every scalar value occurring in the original program. For instance, `g_alpha(1:2)` is associated with the scalar variable `alpha` while `g_vel(1:2,1:3*L)` is associated with `vel(1:3*L)`.

The basic idea behind AD is the fact that the execution of a code is nothing but a sequence of elementary arithmetic operations such as binary addition or intrinsic functions.

The partial derivatives of these elementary functions are known and, following the chain rule of differential calculus, can be combined in a step-wise manner to yield the overall derivative of the entire program. In contrast to approximations obtained from numerical differentiation like divided differences, derivatives computed by AD are free from truncation error. To illustrate AD, consider the following simple code fragment

```
u = z(1) + z(2)
v = v*u
```

that could appear somewhere inside of `TFS`, and keep in mind that the technique is also applicable to larger codes. We introduce a straightforward AD strategy called the *forward mode* by studying the transformation of the sample code fragment. In the forward mode, a gradient object `g_w` is associated to every variable `w` appearing in the original code. This gradient object stores the partial derivatives of the variable `w` with respect to the input variables of interest. In the `TFS` example, the variable `g_w` stores $\partial \mathtt{w}/\partial(\mathtt{alpha},\mathtt{beta})$. Setting a variable that stores the number of scalar variables with respect to which derivatives are computed to `n=2`, the resulting AD-generated code is given by:

```
do i = 1,n
  g_u(i) = g_z(i,1) + g_z(i,2)
enddo
u = z(1) + z(2)
do i = 1,n
  g_v(i) = g_v(i)*u + v*g_u(i)
enddo
v = v*u
```

This code propagates derivatives of intermediate variables with respect to input variables following the control flow of the original program. If the differentiated code `g_TFS` is called with the initializations

```
g_alpha(1)=1 , g_alpha(2)=0 , g_beta(1)=0 , g_beta(2)=1
```

it computes the desired derivatives

$$\begin{aligned}
\mathtt{g_vel(1,1:3*k)} &= \partial \mathtt{vel(1:3*k)}/\partial \mathtt{alpha} \\
\mathtt{g_vel(2,1:3*k)} &= \partial \mathtt{vel(1:3*k)}/\partial \mathtt{beta} \\
\mathtt{g_pre(1,1:k)} &= \partial \mathtt{pre(1:k)}/\partial \mathtt{alpha} \\
\mathtt{g_pre(2,1:k)} &= \partial \mathtt{pre(1:k)}/\partial \mathtt{beta}
\end{aligned}$$

In addition to the forward mode, automatic differentiation provides the *reverse mode* which propagates derivatives of the output variables with respect to intermediate variables by reversing the control flow of the original program.

3 Automatic Differentiation of MPI Functions

Automatic differentiation of the serial 2D TFS code employing version 2 of the AD-tool ADIFOR[10] is reported in previous publications.[11–13] In the present study, we report on automatic differentiation of the MPI-parallelized version of 3D TFS. Besides the extension from two to three space dimensions, the main achievement compared to our previous work is the semi-automatic transformation of the parallel constructs. Previous work

on automatic differentiation of message-passing parallel programs is published in several contributions.[14–20]

The basic point-to-point communication of the MPI interface is a pair of send and receive commands operating between two processes P_1 and P_2. Assume an operation `send(u,P2)` executed on P_1 that sends a variable `u` from P_1 to process P_2. Assume further that `receive(v,P1)` is the corresponding command on P_2 that receives the data obtained from P_1 in a variable `v`. This pair of communication commands can be thought of as copy to and from a virtual data structure `buffer` that is accessible from both processes. That is, in the notation

```
P1:   send( u, P2 )          P2:   receive( v, P1 )
   // buffer = u                   // v = buffer
```

the statements on the left carried out on P_1 are equivalent as are the statements on the right on P_2. Using the interpretation of a global buffer, the differentiated statements in the forward mode can be derived as

```
P1:// g_buffer = g_u         P2:// g_v = g_buffer
      send( g_u, P2 )              receive( g_v, P1 )
```

By reversing the control flow of the original program, the differentiated statements in the reverse mode are given by

```
P1:// a_u = a_u + a_buffer   P2:// a_buffer = a_buffer + a_v
      receive( tmp, P2 )           send( a_v, P1 )
      a_u = a_u + tmp              a_v = 0
```

Reduction operations combine communication and computation. As an example, consider the case where each process P_i contains a scalar value u_i in a variable `u`. Let `reduce(u,v,'sum')` denote the reduction computing the sum $v = \sum_{i=1}^{p} u_i$ over p processes, where the result is stored in a variable `v` on process, say, P_1. In the forward mode, the derivatives are reduced in the same way. That is, the result `g_v` on P_1 is computed by `reduce(g_u,g_v,'sum')` from given distributed values stored in `g_u`. In the reverse mode, a broadcast of `a_v` from P_1 to all other processes is followed by an update of the form `a_u = a_u + a_v` executed on each process. A recipe for AD of a reduction operation computing the product of distributed values is given in a paper by Hovland and Bischof.[15]

Another issue to be specifically addressed by AD for message-passing parallel programs is activity analysis. The purpose of this data-flow analysis is to find out if a variable is active or not. A variable is called active if it depends on the input variables with respect to which derivatives are computed and if it also influences the output variable whose derivatives are computed. In addition to activity analysis of serial programs, there is need to track variable dependencies that pass through send and receive operations. Typically, activity analysis of message-passing parallel programs is carried out conservatively,[19] overestimating the set of active variables which leads to AD-generated programs whose efficiency in terms of storage and computation could be improved. Only recently, a study is concerned with investigating activity analysis for MPI programs more rigorously.[20]

Since ADIFOR 2 is not capable of transforming MPI library calls, the corresponding MPI routines are excluded from the AD process. That is, using ADIFOR 2 the TFS code is automatically differentiated with respect to the angle of attack and the yaw angle, where

the MPI process communication is ignored. However, in order to ensure correct overall forward propagation of the derivatives, the derivative information computed by the MPI processes needs to be exchanged whenever the original function values are exchanged. In essence, our approach combines AD of the serial TFS code in an automated fashion with a manual activity analysis of MPI library calls along the lines discussed in Strout et al.[20]

In TFS the original code contains the following non-blocking *send* operation

```
  call mpi_isend(A, k, mpi_double_precision, dest,
&              tag, mpi_comm_world, request, ierror)
```

which asynchronously sends the data stored in the buffer `A` to a destination process `dest`. This buffer consists of `k` consecutive entries of type `double precision`. Then, the differentiated code produced by the forward mode of AD requires a similar operation for sending the derivative values `g_A` of `A`, where the number of data entries is increased by the number, n, of directional derivatives propagated through the code. The corresponding non-blocking *send* operation reads as follows:

```
  call mpi_isend(g_A, k*n, mpi_double_precision, dest,
&              tag, mpi_comm_world, request, ierror)
```

Recall from the previous section that the dimension of the derivative `g_A` is increased by one, compared to its associated original variable `A`. The leading dimension of `g_A` is therefore given by `n`. In this study we manually inserted the additional MPI calls for sending the derivative values, although in principle they could be automatically generated by the AD tool. In a completely analogous fashion, for every blocking *receive* operation occurring in the original TFS code, receiving `A` from the MPI process `source`

```
  call mpi_recv(A, k, mpi_double_precision, source,
&             tag, mpi_comm_world, status, ierror)
```

we inserted a corresponding MPI call to receive derivative information from other MPI processes:

```
  call mpi_recv(g_A, k*n, mpi_double_precision, source,
&             tag, mpi_comm_world, status, ierror)
```

Finally, we sketch the corresponding AD transformation for the reverse mode. In the reverse mode of AD, for each scalar program variable in the original code, an adjoint of length m is propagated through the differentiated code. Moreover during the reversal of the control flow of the original program the blocking *receive* operation is replaced by a blocking *send* for the adjoint computation:

```
  call mpi_send(a_A, k*m, mpi_double_precision, dest,
&             tag, mpi_comm_world, ierror)
```

Here, the destination process `dest` is the sending process `source` of the message in the corresponding *receive* operation in the original code.

4 Performance Experiments

In the sequel we present results from numerical experiments with TFS and its differentiated version. The corresponding execution time measurements are carried out on

a Sun Fire E2900 equipped with 12 dual-core UltraSPARC IV processors running at 1.2 GHz clock speed. We compute an inviscid three-dimensional flow around the BAC 3-11/RES/30/21 transonic airfoil which is reported in the AGARD advisory report no. 303[21] and used as reference airfoil for the collaborative research center SFB 401. The Mach and Reynolds numbers are $M_\infty = 0.689$ and $Re = 1.969 \times 10^6$, respectively. The angle of attack is -0.335° and the yaw angle is 0.0°. The computational grid for TFS consists of approximately 77,000 grid points and is divided into three different blocks.

The execution times for the parallel code running on one and three processors is given in Table 1. The table also shows the resulting speedup, taking the execution time of the parallel code with one MPI task as reference. The second column refers to the original TFS code while the third column contains the data of its differentiated version denoted by TFS$'$. The ratio of the third to second column is displayed in the fourth column of this table.

Metric	TFS	TFS$'$	Ratio
Serial execution time [s]	338	2197	6.50
Parallel execution time [s]	157	969	6.17
Speedup	2.15	2.27	1.05

Table 1. Execution times required for performing 100 iterations with TFS and its differentiated version TFS$'$. The serial and parallel execution time is measured on a Sun Fire E2900 using one and three processors, respectively.

While the original TFS code achieves a speedup of 2.15 employing three MPI tasks, TFS$'$ yields a slightly better speedup of 2.27 for the same configuration. However, optimal speedup cannot be achieved because the three processors are assigned to blocks of different sizes. While the largest of the three blocks consists of 35,640 grid points, the remaining two blocks comprise 24,687 and 17,091 grid points, respectively. This leads to a work load imbalance where one task is assigned about 46% of the total amount of computational work, while the other two tasks perform only 32% and 22% of the computational work, respectively. As a consequence two of the three MPI processes spend a certain amount of time waiting for the process with the higher work load, assuming identical processor types. A run-time analysis of the execution trace using the VAMPIR[22] performance analysis and visualization tool reveals that about 23.7% of the overall execution time is spent in MPI communication.

5 Concluding Remarks

Sensitivities of selected output variables with respect to selected input variables of large-scale simulation codes are ubiquitous in various areas of computational science and engineering. Examples include optimization, solution of nonlinear systems of equations, or inverse problems. Rather than relying on numerical differentiation based on some form of divided differencing that inherently involves truncation error, we suggest the use of techniques of automatic differentiation. While this program transformation applied to serial programs has proven to work in a robust and stable fashion under a wide range of

circumstances, no current automatic differentiation tool is capable of handling message-passing parallel programs in a fully automated way. In fact, the number of successful transformation for non-trivial parallel codes published in the open literature is small. For a concrete three-dimensional computational fluid dynamics application arising from aerodynamics, we successfully applied automatic differentiation to an MPI-parallelized finite volume code solving the Navier–Stokes equations. More precisely, we computed the sensitivities of the flow field around a transonic airfoil with respect to the angle of attack and the yaw angle. Though most of the program transformation is handled automatically, the differentiation of the MPI functions is carried out manually.

Acknowledgements

We appreciate the comments by the anonymous reviewers and would like to thank the Institute of Aerodynamics for making available the source code of the flow solver TFS. This research is partially supported by the Deutsche Forschungsgemeinschaft (DFG) within SFB 401 "Modulation of flow and fluid–structure interaction at airplane wings," RWTH Aachen University, Germany. The Aachen Institute for Advanced Study in Computational Engineering Science (AICES) provides a stimulating research environment for our work.

References

1. D. Hänel, M. Meinke, and W. Schröder, *Application of the multigrid method in solutions of the compressible Navier–Stokes equations*, in: Proc. 4th Copper Mountain Conference on Multigrid Methods, J. Mandel, (Ed.), pp. 234–254, SIAM, Philadelphia, PA, (1989).
2. M. Meinke and E. Krause, *Simulation of incompressible and compressible flows on vector-parallel computers*, in: Proceedings of the 1998 Parallel CFD Conference, Hsinchu, Taiwan, May 11–14, 1998, pp. 25–34, (1999).
3. M. S. Liou and Ch. J. Steffen, *A new flux splitting scheme*, J. Comp. Phys., **107**, 23–39, (1993).
4. M. S. Liou, *A sequel to AUSM: AUSM*$^+$, J. Comp. Phys., **129**, 164–182, (1996).
5. C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and E. Slusanschi, *Efficient and accurate derivatives for a software process chain in airfoil shape optimization*, Future Generation Computer Systems, **21**, 1333–1344, (2005).
6. C. H. Bischof, H. M. Bücker, and A. Rasch, *Sensitivity analysis of turbulence models using automatic differentiation*, SIAM Journal on Scientific Computing, **26**, 510–522, (2005).
7. L. B. Rall, *Automatic differentiation: techniques and applications*, vol. **120** of *Lecture Notes in Computer Science*, (Springer Verlag, Berlin, 1981).
8. A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, (SIAM, Philadelphia, 2000).
9. P. Heimbach and C. Hill and R. Giering, *An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation*, Future Generation Computer Systems, **21**, 1356–1371, (2005).

10. C. Bischof, A. Carle, P. Khademi, and A. Mauer, *Adifor 2.0: Automatic differentiation of Fortran 77 programs*, IEEE Computational Science & Engineering, **3**, 18–32, (1996).
11. H. M. Bücker, B. Lang, A. Rasch, and C. H. Bischof, *Sensitivity analysis of an airfoil simulation using automatic differentiation*, in: Proc. IASTED International Conference on Modelling, Identification, and Control, Innsbruck, Austria, February 18–21, 2002, M. H. Hamza, (Ed.), pp. 73–76, (ACTA Press, Anaheim, CA, 2002).
12. H. M. Bücker, B. Lang, A. Rasch, and C. H. Bischof, *Computation of sensitivity information for aircraft design by automatic differentiation*, in: Computational Science – ICCS 2002, Proc. International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II, P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, (Eds.), vol. **2330** of *Lecture Notes in Computer Science*, pp. 1069–1076, (Springer, Berlin, 2002).
13. C. H. Bischof, H. M. Bücker, B. Lang, and A. Rasch, "Automated gradient calculation", in: Flow Modulation and Fluid-Structure Interaction at Airplane Wings, J. Ballmann, (Ed.), number 84 in Notes on Numerical Fluid Mechanics and Multidisciplinary Design, pp. 205–224, (Springer, Berlin, 2003).
14. P. D. Hovland, *Automatic differentiation of parallel programs*, PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, (1997).
15. P. D. Hovland and C. H. Bischof, *Automatic differentiation for message-passing parallel programs*, in: Proc. First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, pp. 98–104, IEEE Computer Society Press, Los Alamitos, CA, USA, (1998).
16. C. Faure and P. Dutto, *Extension of Odyssée to the MPI library – Direct mode*, Rapport de recherche 3715, INRIA, Sophia Antipolis, (1999).
17. C. Faure and P. Dutto, *Extension of Odyssée to the MPI library – Reverse mode*, Rapport de recherche 3774, INRIA, Sophia Antipolis, (1999).
18. A. Carle and M. Fagan, *Automatically differentiating MPI-1 datatypes: The complete story*, in: Automatic Differentiation of Algorithms: From Simulation to Optimization, G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, (Eds.), pp. 215–222, (Springer, New York, 2002).
19. A. Carle, *Automatic Differentiation*, in: Sourcebook of Parallel Computing, J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennnedy, L. Torczon, and A. White, (Eds.), pp. 701–719, (Morgan Kaufmann Publishers, San Francisco, 2003).
20. M. Mills Strout, B. Kreaseck, and P. D. Hovland, *Data-Flow Analysis for MPI Programs*, in: Proceedings of the International Conference on Parallel Processing (ICPP-06), Columbus, Ohio, USA, August 14–18, 2006, pp. 175–184, IEEE Computer Society, Los Alamitos, CA, USA, (2006).
21. I. R. M. Moir, *Measurements on a two-dimensional aerofoil with high-lift devices*, in: AGARD–AR–303: A Selection of Experimental Test Cases for the Validation of CFD Codes, vol. 1 and 2. Advisory Group for Aerospace Research & Development, Neuilly-sur-Seine, France, (1994).
22. W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, *VAMPIR: Visualization and Analysis of MPI Resources*, Supercomputer, **12**, no. 1, 69–80, (1996).